Slide 1

Partie 2 Programmation distribuée

Modèle de programmation concurrente vu jusqu'à présent dans le livre d'Andrews = communication et synchronisation par variables partagées

Autre modèle possible = communication et synchronisation par *échange de messages*

$\Rightarrow$ chaque processus possède sa propre mémoire privée, inaccessible aux autres processus

$\Rightarrow$ pas de notion d'exclusion mutuelle ou de section critique

Notion clé = *canal* de communication

$\Rightarrow$ définit un chemin (un lien) de communication entre processus

Slide 2

Styles de programmation concurrente par échanges de messages:

- Définition statique des canaux vs. dynamique:
  - Statique: un canal est créé de façon statique et établit un lien de façon directe et permanente entre deux processus
  - Dynamique: de nouveaux canaux peuvent être créés en cours d'exécution et peuvent lier différents processus
- Envoi synchrone vs. asynchrone:
  - Synchrone = Les deux processus doivent être prêts à communiquer pour que l'échange ait lieu, sinon le premier attend
  - Asynchrone = Aucun blocage de l'envoyeur, même si le récepteur n'est pas prêt
- Communication uni-directionnelle ou bi-directionnelle:
  - Uni-directionnelle: Transmission d'un émetteur vers un récepteur
  - Bi-directionnelle: Échange symétrique d'information entre les deux processus

Slide 3

7 Échange de messages

Dans ce chapitre (sauf Section 7.5): style *asynchrone* d'envoi de messages

Deux primitives:

1. `send`: envoie un message sur un canal
2. `receive`: reçoit un message sur un canal

Analogie entre ces primitives et celles des sémaphores (lorsque les messages envoyés sont vides):

1. `send` = `V`
2. `receive` = `P`

Slide 4

7.1 Échange asynchrone de messages

Ce que contient un *canal* de communication:

- canal = *file* (*queue*) des messages qui ont été envoyés mais qui n'ont pas encore été reçus

Déclaration d'un canal:

```
chan ch( type_1 id_1, ..., type_n id_n );
```

Interprétation:

- Nom du canal = `ch`
- Types des éléments transmis dans les messages = `type_i`
- Les identificateurs `id_i` sont optionnels

Exemples:

```
chan input(char);
chan acces_disque(int cylindre, int bloc,
                  int nb_octets, char *tampon);
chan result[n](int);
```

Slide 5

Envoi d'un message (les expressions `expr_i` doivent évidemment être du bon type):

```
send ch(expr_1, ..., expr_n);
```

Sémantique:

1. Les expressions sont évaluées, un message est créé et est ajouté dans la file associée au canal `ch`
2. L'opération `send` ne bloque pas, c'est-à-dire, se termine immédiatement: on suppose que les files sont non-bornées

Réception d'un message (les `var_i` doivent être du bon type):

```
receive ch(var_1, ..., var_n);
```

Sémantique:

1. Le receveur est *bloqué* jusqu'à ce qu'un message soit disponible
2. Le message à la tête de la file associée au canal est retiré de la file et les champs du message sont associés aux différentes variables `var_i`

Slide 6

Exemple: un filtre pour assembler des caractères en lignes complètes = Figure 7.1 (p. 297)

- Les canaux sont déclarés au niveau global de façon à ce qu'ils soient connus de tous les processus qui les utiliseront
- Le processus reçoit des `char` en entrée, puis envoie des lignes en sortie (type `char[MAXLINE]`)

Différents styles de canaux:

- *Boîte aux lettres* = canal sur lequel n'importe quel processus peut envoyer des messages ou à partir duquel n'importe quel processus peut en recevoir
- *Port d'entrée* (*input port*) = canal sur lequel n'importe quel processus peut envoyer des messages mais un seul processus en reçoit
- *Lien entre processus* (*link*) = canal utilisé par un seul émetteur et un seul récepteur

Slide 7

7.2 Filtres: Un réseau pour trier

Filtre = processus qui reçoit des messages sur un ou plusieurs canaux d'entrée et qui envoit des messages sur un ou plusieurs canaux de sortie

Un filtre pour fusionner le contenu des messages provenant de deux canaux: Figure 7.2 (p. 300)

Un réseau de filtres pour trier `n` nombres: Figure 7.3 (p. 301)

- Contient `n-1` processus
- Profondeur log_2 `n`
- Nombre total de canaux = `2n-1`

Slide 8

Deux approches pour créer le réseau, de façon à que les différents processus soient correctement reliés entre eux par des canaux:

1. Approche statique: Les canaux sont déclarés dans un tableau global et on inscrit (*embed*) l'arbre des canaux dans le tableau de façon à ce que le processus `i` puisse accéder aux canaux appropriés à partir de son index `i`.
2. Approche dynamique: Les canaux sont déclarés de façon globale et chaque instance du processus `Merge` reçoit les 3 canaux appropriés au moment où l'instance est créée $\Rightarrow$ on doit avoir un processus maître qui crée les processus et leur passe les arguments appropriés

7.3 Clients et serveurs

Serveur = de façon répétitive, recoît des requêtes et les traite

Client = envoie une requête à un serveur puis attend la réponse

7.3.1 Moniteurs actifs

Moniteur ≈ serveur: reçoit des requêtes pour manipuler, de façon exclusive et atomique, une ressource partagée encapsulée par le moniteur

Slide 9

Moniteur simplifié = avec une seule opération:

```
monitor Mname {
  declaration des variables permanentes;

  code pour initialisation;

  procedure op( parametres formels ) {
    corps de l'operation op;
  }
}
```

Simulation du moniteur `Mname` à l'aide d'un processus serveur:

- Les variables permanentes du moniteur deviennent les variables privées du processus
- Le code d'initialisation du moniteur est réalisé par le segment initial du processus
- Une exécution de l'opération `op` est réalisée en envoyant un message approprié sur un canal. Le processus serveur fait une boucle pour recevoir ces requêtes sur le canal puis les traite.

Slide 10

Donc:

- Comme la mise en oeuvre des *split-phase locks* en Threaded-C (`LOCK.[hc]`)
- Comme la mise en oeuvre des sémaphores en Threaded-C (`SEMAPHORE.[hc]`)

Slide 11

Exemple: Moniteur avec une seule opération — c'est-à-dire, on suppose que l'opération reçoit k arguments et retourne un seul résultat.

```
chan requete( int clientID, t_1 a_1, ..., t_k a_k );
chan reponse[n]( t_r rep );

process Serveur {
   int clientID;
   t_1 v_1; ...; t_k v_k;
   t_r rep;
   declaration des variables permanentes;

   code pour initialisation des variables permanentes;

   while( TRUE ) {
     receive requete(clientID, v_1, ..., v_k);
     code pour realiser l'operation op;
     send reply[clientID](rep);
   }
 }
```

Slide 12

Généralisation pour plusieurs opérations: types `union` pour les messages et type énuméré pour identifier l'opération désirée

```
type sorte_op = enum(OP_1, ..., OP_n);
type arg_type = union(arg_1, ..., arg_n);
type res_type = union(res_1, ..., res_n);

chan requete(int clientID, sorte_op, arg_type);
chan reponse[n](res_type);

process Serveur {
  ...
  while(TRUE) {
    receive requete(clientID, sorte, args);
    switch(sorte) {
       case OP_1: ...; break;
       ...
    }
    send reply[clientID](resultats);
  }
}
```

Slide 13

7.4 *Interacting peers*: Échange de valeurs

Problème = On a `n` processus, où chacun conserve une valeur locale (nombre entier). L'objectif est de faire en sorte que chaque processus, à la fin du programme, connaisse la valeur minimum *et* la valeur maximum parmi toutes les valeurs.

Trois solutions possibles:

1. Solution centralisée
2. Solution symétrique (SPMD)
3. Solution quasi-symétrique avec anneau de processus

Voir Figure 7.14 (p. 317)

Slide 14

1. Solution centralisée: Figure 7.11 (p. 315) = toutes les valeurs sont envoyées à un processus coordonnateur. Ce dernier détermine les valeurs min et max, puis les transmets aux autres processus.

   Caractéristiques:
   - Requiert `2(n-1)` messages.
   - Requiert un processus qui fait tout le travail

2. Solution symétrique: Figure 7.12 (p. 315) = chaque processus envoie sa valeur à tous les autres processus.

   Caractéristiques:
   - Solution SPMD (*Single Program Multiple Data*): tous les processus exécutent exactement le même code
   - Requiert `n(n-1)` messages.

Slide 15

3. Solution avec processus organisés en anneau: Figure 7.13 (p. 317) = une première série de messages permet au dernier processus de déterminer le min et le max, puis une deuxième série de messages permet de transmettre ces min et max à l'ensemble des autres processus.

   Caractéristiques:

   - Solution quasi-symétrique (seul `P[0]` est légèrement différent)
   - Requiert $2n$ messages.

Slide 16

7.5 Envoi synchrone de messages

Envoi synchrone de messages = `sync_send` $\Rightarrow$ l'envoyeur bloque jusqu'à ce que le récepteur reçoive le message

Avantages:

- La taille des queues associées aux canaux peut être bornée $\Rightarrow$ un espace borné (alloué de façon statique) peut être utilisé pour mettre en oeuvre les canaux

Désavantages:

- Moins de concurrence: pour chaque échange de message, on aura nécessairement qu'un des deux processus impliqués bloquera
- Plus difficile à programmer car il y a danger de générer des *deadlocks*

Slide 17

Exemple avec *deadlock*:

```
chan in1(int), in2(int);

process P1 {
  int value1 = 1, value2;

  sync_send in2(value1);
  receive in1(value2);
}

process P2 {
  int value, value2 = 2;

  sync_send in1(value2);
  receive in2(value1);
}
```